Scalable Frequent Sequence Mining With Flexible Subsequence Constraints

Alexander Renz-Wieland
Technische Universität Berlin
alexander.renz-wieland@tu-berlin.de

Matthias Bertsch
Universität Mannheim
mabertsc@mail.uni-mannheim.de

Rainer Gemulla
Universität Mannheim
rgemulla@uni-mannheim.de

***Abstract*—We study scalable algorithms for frequent sequence mining under flexible subsequence constraints. Such constraints enable applications to specify concisely which patterns are of interest and which are not. We focus on the bulk synchronous parallel model with one round of communication; this model is suitable for platforms such as MapReduce or Spark. We derive a general framework for frequent sequence mining under this model and propose the D-SEQ and D-CAND algorithms within this framework. The algorithms differ in what data are communicated and how computation is split up among workers. To the best of our knowledge, D-SEQ and D-CAND are the first scalable algorithms for frequent sequence mining with flexible constraints. We conducted an experimental study on multiple real-world datasets that suggests that our algorithms scale nearly linearly, outperform common baselines, and offer acceptable generalization overhead over existing, less general mining algorithms.**

I. Introduction

Frequent sequence mining (FSM) is a data mining task that finds frequent subsequences in a sequence database. FSM is ubiquitous in applications, including natural language processing [19], information extraction [12], web usage mining [29], market-basket analysis [28], and computational biology [9].

Fig. 1 gives an overview of prior FSM algorithms, categorized along the dimensions of flexibility and scalability. Roughly speaking, more flexible algorithms aim to support a wider range of applications, whereas scalable algorithms can handle very large datasets with hundreds of millions of sequences. Prior work on FSM focused mostly on one of the dimensions. More specifically, a number of scalable FSM algorithms has been proposed [6], [8], [15], [16], [21], [22], [35]. These algorithms are inflexible, however, in that they cannot be tailored to a particular application. They often produce a multitude of frequent subsequences, among only few may be interesting to applications [27]. One approach to improve flexibility is the use of subsequence constraints, which specify conditions under which a subsequence is potentially interesting to the particular application. Ordered by increasing flexibility, common types of subsequence constraints include length constraints [28], [34], gap and duration constraints [14], [28], [34], hierarchy constraints [28], "output filter" regular expression constraints [2], [3], [13], [31], and regular expression constraints with capture groups and hierarchies [5], [7]. The latter type subsumes the remaining ones, and we subsequently refer to it as *flexible constraints*.

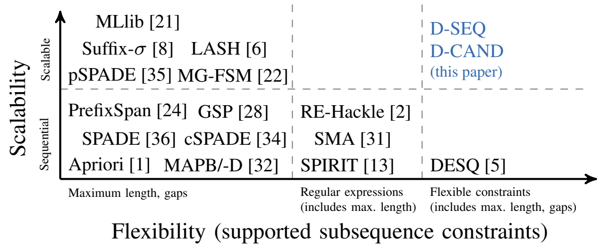


Fig. 1: Selected FSM algorithms, arranged by scalability and flexibility in terms of supported subsequence constraints.

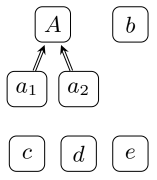
Consider for example the task of mining frequent relational phrases between entities from large text corpora as in [12], [23]; e.g., the phrase *make a deal with* may be frequent between persons and/or organizations. An FSM algorithm that does not support flexible constraints cannot solve such a task: it cannot be tailored to consider only relational phrases, thereby producing many uninteresting (i.e., non-relational) patterns, and it does not support context constraints, thereby producing spurious patterns (i.e., patterns that do not connect entities). In contrast, FSM algorithms that support flexible constraints can express this task—e.g., using a constraint such as ENTITY (VERB$^+$ DET? NOUN$^+$? PREP?) ENTITY [12] in the pattern language of DESQ [5], [7]—but they cannot handle very large datasets. Other examples involving flexible constraints include the construction of the well-known Google n-gram corpus [30] and mining of protein sequences that exhibit a given motif [31].

In this paper, we study FSM algorithms that are both flexible and scalable. We focus on the bulk synchronous parallel model with one round of communication, which is suitable for platforms such as MapReduce or Spark. We propose the D-SEQ and D-CAND algorithms, which differ in how work is distributed and what data are communicated among workers. More specifically, we make the following contributions:

- We generalize existing approaches for distributed FSM with one round of communication to a general framework that supports flexible subsequence constraints (Sec. III).
- We propose D-SEQ (Sec. V), an FSM algorithm that communicates rewritten input sequences [6], [22] among workers. The algorithm provides robust performance across a wide range of subsequence constraints.

T_1: a_1cdcb
T_2: eea_1ea_1eb
T_3: $cdcb$
T_4: a_2db
T_5: a_1a_1b

(a) Sequence db.



(b) Item hierarchy

w	$f(w, \mathscr{D}_{\text{ex}})$
b	5
A	4
d	3
a_1	3
c	2
e	1
a_2	1

(c) Item freq.

Fig. 2: Running example. Sequence database $\mathscr{D}_{\text{ex}}$, item hierarchy, and item frequencies $f(w, \mathscr{D}_{\text{ex}})$.

- We propose D-CAND (Sec. VI), an FSM algorithm that communicates candidates (in compressed form). The algorithm is tailored to more selective constraints and it mines such constraints more efficiently than D-SEQ.
- We report results of an experimental study (Sec. VII) that examines the relative performance of D-SEQ and D-CAND and compares them to baseline and state-of-the-art (specialized) methods on real-world datasets.

We found that our algorithms scaled nearly linearly with the number of sequences in the sequence database. They also had acceptable generalization overhead (between 0.9x and 4.3x) over existing specialized methods, which cannot handle flexible constraints.

II. Preliminaries

We start with introducing basic concepts and a formal definition of the FSM problem that we consider in this paper.

Sequence database. A *sequence database* is a set[1] of sequences, denoted $\mathscr{D} = \{ T_1, T_2, \ldots, T_{|\mathscr{D}|} \}$. Each *sequence* $T = t_1 t_2 \ldots t_{|T|}$ is a list of items from a *vocabulary* $\Sigma = \{ w_1, w_2, \ldots, w_{|\Sigma|} \}$. We denote by ϵ the empty sequence, by $|T|$ the length of sequence T, by Σ^* the set of all sequences that can be constructed from items in Σ. Fig. 2a shows an example sequence database $\mathscr{D}_{\text{ex}}$ with 5 sequences.

Item hierarchy. The items in Σ are arranged in an *item hierarchy*, i. e., a directed acyclic graph that expresses how items can be generalized (or that they cannot be generalized). With item hierarchies, analysts can succinctly express constraints or find patterns involving general concepts that may not occur directly in the data (e.g., $make$ may generalize to $VERB$). Fig. 2b shows an example hierarchy in which, for example, item a_1 generalizes to item A. We say that an item u *generalizes directly to* an item v, denoted $u \Rightarrow v$, if u is a child of v. We further denote by $\Rightarrow^*$ the reflexive transitive closure of $\Rightarrow$. For each item $w \in \Sigma$, we denote by $\text{anc}(w) = \{ w' \mid w \Rightarrow^* w' \}$ the set of *ancestors* of w (including w) and by $\text{desc}(w) = \{ w' \mid w' \Rightarrow^* w \}$ the set of *descendants* of w (again, including w). In our running example, we have $\text{anc}(a_1) = \{ a_1, A \}$ and $\text{desc}(A) = \{ A, a_1, a_2 \}$.

Subsequence. Let $S = s_1 s_2 \ldots s_{|S|}$ and $T = t_1 t_2 \ldots t_{|T|}$ be two sequences composed of items from Σ. We say that S is a *subsequence* of T, denoted $S \sqsubseteq T$, if S can be obtained by deleting and/or generalizing items in T. More formally, $S \sqsubseteq T$ if and only if there exist integers $1 \leq i_1 < i_2 < \cdots < i_{|S|} \leq |T|$ such that $t_{i_j} \Rightarrow^* s_j$ for $1 \leq j \leq |S|$. Continuing our example, we have $a_1a_1b \sqsubseteq T_5$ and $Ab \sqsubseteq T_5$, but $a_1e \not\sqsubseteq T_5$.

[1]To simplify exposition, we assume that input sequences are distinct.

TABLE I: Selected pattern expressions. Pattern expression E matches any item $t \in \text{in}_E$ and outputs any element of $\text{out}_E(t)$.

E	in_E	$\text{out}_E(t)$	Description
$.$	$t \in \Sigma$	$\{ \epsilon \}$	Match any item, empty output
$(.^{\uparrow})$	$t \in \Sigma$	$\text{anc}(t)$	Match any item, output ancestors
(w)	$t \in \text{desc}(w)$	$\{ t \}$	Match any desc. of w, output matched item

Subsequence constraints. We follow [5], [7] and express subsequence constraints using *subsequence predicates* of form $\pi : \Sigma^* \times \Sigma^* \to \{ 0, 1 \}$. We say that S is a *π-subsequence* of T, denoted $S \sqsubseteq_\pi T$, if $S \sqsubseteq T$ and $\pi(S,T) = 1$. We then also say that T *π-generates* S. Denote by

$$G_\pi(T) = \{ S \mid S \sqsubseteq_\pi T \}$$

the set of subsequences π-generated by T. The subsequences in $G_\pi(T)$ constitute *candidate subsequences* for FSM. For the example of Fig. 2, a subsequence predicate π_{ex} may specify that we are interested in only the subsequences that begin with A or one of its descendants and end with b. Adopting the the language of [5], [7] (see below), we can express this constraint using *pattern expression*

$$\pi_{\text{ex}} = .^*(A)[(.^{\uparrow}).^*]^*(b).^*$$

We have $G_{\pi_{\text{ex}}}(T_5) = \{ a_1a_1b, a_1Ab, a_1b \}$. Note that, for example, $b \sqsubseteq T_5$ but $b \not\sqsubseteq_{\pi_{\text{ex}}} T_5$. Fig. 3 depicts the candidate subsequences for all $T \in \mathscr{D}_{\text{ex}}$.

Pattern expression language. The pattern expression language is defined inductively: (1) For each item $w \in \Sigma$, the expressions w, $w_=$, $w^{\uparrow}$, and $w^{\uparrow}_=$ are pattern expressions. (2) $.$ and $.^{\uparrow}$ are pattern expressions. (3) If E is a pattern expression, so are (E), $[E]$, $[E]^*$, $[E]^+$, $[E]?$, and for all $n, m \in \mathbb{N}$ with $n \leq m$, $[E]\{n\}$, $[E]\{n,\}$, and $[E]\{n,m\}$. (4) If E_1 and E_2 are pattern expressions, so are $[E_1E_2]$ and $[E_1|E_2]$.

Pattern expressions are based on regular expressions, but additionally include capture groups (in parentheses), hierarchies (by omitting $_=$), and generalizations (using $^{\uparrow}$ and $^{\uparrow}_=$). Intuitively, pattern expressions work like regular expressions: when they match, they output what is captured and may generalize along the hierarchy (optionally via $^{\uparrow}$, always via $^{\uparrow}_=$). The language makes use of the usual precedence rules for regular expressions to suppress square brackets (but not parentheses); operators that appear earlier in the above definition have higher precedence. Tab. I shows the input and output of selected pattern expressions. For example, $Aa_1b \not\sqsubseteq_{\pi_{\text{ex}}} T_5$, because pattern expression (A) does not allow to generalize matched items, i.e., $\text{out}_{(A)}(a_1) = \{ a_1 \}$.

Tab. III (page 9) gives examples of application pattern expressions. A more detailed description of the syntax and semantics of pattern expressions can be found in [7].

Support. The support of a subsequence S in a sequence database $\mathscr{D}$ is the set of input sequences that π-generate S:

$$\text{Sup}_\pi(S, \mathscr{D}) = \{ T \in \mathscr{D} \mid S \in G_\pi(T) \} .$$

Algorithm 1: Distributed FSM in MapReduce

```
   Data: Database $\mathscr{D}$, constraint $\pi$, threshold $\sigma$
1  Function Map($T$)                          // process input sequence $T$
2      $K(T) \leftarrow$ keys of partitions for which $T$ is relevant
3      foreach $k \in K(T)$ do
4          $\rho_k(T) \leftarrow$ representation to send to partition $\mathcal{P}_k$
5          Emit $\langle k, \rho_k(T) \rangle$

6  Function Reduce($k$, $\mathcal{P}_k$)                 // process partition $\mathcal{P}_k$
7      $F_k(\mathcal{P}_k) \leftarrow$ candidate subsequences with partition key $k$
         along with their frequencies
8      foreach $(S, f) \in F_k(\mathcal{P}_k)$ do
9          if $f \geq \sigma$ then
10             Emit $\langle S, f \rangle$
```

Denote by $f_\pi(S, \mathscr{D}) = |\mathrm{Sup}_\pi(S, \mathscr{D})|$ the *frequency* of S in $\mathscr{D}$. Given a *minimum support threshold* $\sigma > 0$, a subsequence S is *frequent* if $f_\pi(S, \mathscr{D}) \geq \sigma$.

We sometimes omit π to refer to the unconstrained support or frequency. For example, $f(w, \mathscr{D})$ refers to the *item frequency* of w in $\mathscr{D}$; see Fig. 2c. The set of all *frequent items* along with their frequency is called the *f-list*. We assume throughout that the f-list is known.

Problem statement. Given a sequence database $\mathscr{D}$, a subsequence predicate π, and a minimum support threshold σ, output each frequent subsequence (w.r.t. π and σ) along with its frequency.

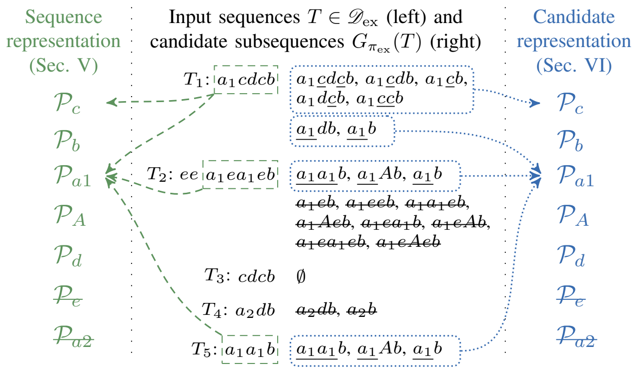
For π_{ex} and $\sigma = 2$ in $\mathscr{D}_{\text{ex}}$, we find frequent subsequences a_1a_1b and a_1Ab with frequency 2 and a_1b with frequency 3.

III. Overview

We first discuss a framework for distributed FSM with flexible constraints and one round of communication. The framework generalizes prior distributed algorithms [6], [22].

We assume throughout that the sequence database is distributed across a set of machines, each holding a subset of the input sequences. We focus on algorithms that operate in three phases: (1) process each input sequence independently (*map*), (2) construct a set of partitions (*shuffle*), and (3) mine each resulting partition independently (*reduce*). We require for correctness that each frequent subsequence is output exactly once and with its correct frequency (and no other subsequences are output). Parallel computation is performed in the map and reduce phases, communication in the shuffle phase. Such algorithms fit the bulk synchronous parallel model (with one round of communication) and are suitable for platforms such as MapReduce [11] or Spark [33]. Alg. 1 shows a basic MapReduce implementation that any such algorithm follows.

As each frequent subsequence needs to be output exactly once, Alg. 1 implicitly partitions the space of subsequences: each subsequence S is associated with exactly one of the partitions (the one that ultimately outputs S in case it is frequent). To model this property, we associate with each partition $\mathcal{P}_k$ a unique *partition key* k—which we write as subscript—and with each sequence S the key $\kappa(S)$ of the partition associated with S. Partition $\mathcal{P}_k$ then outputs S if and only if $\kappa(S) = k$ and S is frequent.



Sequence representation (Sec. V)
Input sequences $T \in \mathscr{D}_{\text{ex}}$ (left) and candidate subsequences $G_{\pi_{\text{ex}}}(T)$ (right)
Candidate representation (Sec. VI)
$\mathcal{P}_c$
$\mathcal{P}_b$
$\mathcal{P}_{a1}$
$\mathcal{P}_A$
$\mathcal{P}_d$
$\mathcal{P}_e$
$\mathcal{P}_{a2}$
T_1: a_1cdcb
$a_1\underline{c}d\underline{c}b$, $a_1\underline{c}db$, $a_1\underline{c}b$, $a_1d\underline{c}b$, $a_1\underline{cc}b$
$\underline{a_1}db$, $\underline{a_1}b$
T_2: $ee\,a_1ea_1eb$
$\underline{a_1a_1}b$, $\underline{a_1}Ab$, $\underline{a_1}b$
~~a_1eb~~, ~~a_1eeb~~, ~~a_1a_1eb~~, ~~a_1Aeb~~, ~~a_1ea_1b~~, ~~a_1eAb~~, ~~a_1ea_1eb~~, ~~a_1eAeb~~
T_3: $cdcb$
$\emptyset$
T_4: a_2db
~~a_2db~~, ~~a_2b~~
T_5: a_1a_1b
$\underline{a_1a_1}b$, $\underline{a_1}Ab$, $\underline{a_1}b$

Fig. 3: Item-based partitioning for the running example ($\sigma = 2$). Pivot items are underlined. Candidate subsequences that contain infrequent items and partitions of infrequent items are crossed out.

Every input sequence T carries information that may be relevant for some of the partitions. Roughly speaking, if T contains a candidate subsequence S—i.e., $S \in G_\pi(T)$—then T is potentially relevant for partition $\mathcal{P}_{\kappa(S)}$. Likewise, if T does not contain any subsequence associated with some partition $\mathcal{P}_k$, T is not relevant for $\mathcal{P}_k$. When processing T, Alg. 1 (line 2) first determines the set $K(T)$ of (the keys of) the partitions which "need to know" about T. For each such partition $\mathcal{P}_k$, Alg. 1 (line 4) constructs a *representation* $\rho_k(T)$ that contains the required information for partition $\mathcal{P}_k$, potentially in compressed form. Partition $\mathcal{P}_k$ then collects the representations obtained from all input sequences and determines the subset $F_k(\mathcal{P}_k)$ of all frequent subsequences S with $\kappa(S) = k$ in a *local mining* phase (line 7).

Different distributed FSM algorithms differ in the definition of $K(T)$, $\rho_k(T)$, and $F_k(\mathcal{P}_k)$. The key challenge is to simultaneously ensure correctness, efficiency, and scalability.

A. Subsequence-Based Partitioning

A NAÏVE approach to distributed FSM is to generate all candidate subsequences and subsequently count their frequency (similar to word count). This approach corresponds to a *subsequence-based partitioning*. In our notation, NAÏVE sets $\kappa^{\text{sp}}(S) = S$ (and, consequently, $K(T) = G_\pi(T)$), $\rho_k(T) = 1$, and $F_k(\mathcal{P}_k) = \{\, (k, |\mathcal{P}_k|) \,\}$.

This naïve approach is simple. It is also efficient if $G_\pi(T)$ is small on average—that is, if each input sequence generates few candidates. For subsequence predicates that generate many candidates, NAÏVE is often infeasible: in the worst case, the number of candidate subsequences for a given input sequence is exponential in the sequence length. Another issue with NAÏVE is that partition sizes may not be balanced: partitions corresponding to frequent subsequences are significantly larger than those corresponding to infrequent ones.

A restricted form of *support antimonotonicity* holds in the context of subsequence predicates [5], [7]: for all $w \in S$ and for all π, $f(w, \mathscr{D}) \geq f_\pi(S, \mathscr{D})$. Consequently, no frequent subsequence can contain an infrequent item. Denote by

$$G_\pi^\sigma(T) = \{ S \sqsubseteq_\pi T \mid \forall w \in S : f(w, \mathscr{D}) \geq \sigma \}$$

the set of candidate subsequences that consist only of frequent items. SEMI-NAÏVE, an improved version of NAÏVE, sets $K(T) = G_\pi^\sigma(T)$ (and is otherwise equivalent to NAÏVE), thereby constructing partitions only for candidate subsequences that consist entirely of frequent items. When there are many infrequent items, the SEMI-NAÏVE algorithm can be significantly more efficient than the NAÏVE algorithm; the worst-case behavior remains unaffected.

B. Item-Based Partitioning

Item-based partitioning [6], [10], [17], [22] prevents the exponential number of partitions that can arise in subsequence-based partitioning. Each partition $\mathcal{P}_k$ is responsible for a single item $k \in \Sigma$. Consequently, there are at most $|\Sigma|$ partitions. Subsequence S is associated with partition

$$\kappa^{\mathrm{ip}}(S) = \max \{ w \in S \},$$

where the maximum is taken w.r.t. some total order $<$ on the items in the vocabulary. Following [22], we subsequently refer to the partitioning key in item-based partitioning $\kappa^{\mathrm{ip}}(S)$ as the *pivot item* of S (similarly, to k as the pivot item of $\mathcal{P}_k$), and to a subsequence S with $k = \kappa^{\mathrm{ip}}(S)$ as a *pivot sequence* for pivot item k. Thus, partition $\mathcal{P}_k$ is responsible for mining all frequent subsequences that contain k but no item larger than k or, equivalently, all frequent pivot sequences for k.

The total order $<$ has a large impact on the balance of partition sizes. A common approach is to define $<$ such that $w_1 < w_2$ if $f(w_1, \mathscr{D}) > f(w_2, \mathscr{D})$. Then the pivot item of a sequence is its least frequent item. For example, Fig. 2c shows items $w \in \mathscr{D}_{\mathrm{ex}}$ in such an order: $b < A < \cdots < a_2$. The reasoning behind this approach is that frequent items occur in many input sequences, but their partitions are responsible for few distinct subsequences. For example, for the most frequent item b, partition $\mathcal{P}_b$ outputs only sequences of form b, bb, bbb, and so on. Beedkar and Gemulla [4] show that when the representation $\rho_k(T)$ is constructed appropriately (see below), little information needs to be sent to partitions corresponding to frequent items, which tends to lead to well-balanced partition sizes.

Sequence representation. The key questions in item-based partitioning are how to split up work and how to minimize communication via suitable representations. One option, to which we refer as *sequence representation* and which D-SEQ uses, is to send input sequence T (or an equivalent rewritten variant T') to the partitions for which T is relevant. We set

$$K^{\mathrm{ip}}(T) = \{ k \mid S \in G_\pi^\sigma(T),\, \kappa^{\mathrm{ip}}(S) = k \} \tag{1}$$

and $\rho_k(T) = T$ (or T'). Then, for each subsequence S with $\kappa^{\mathrm{ip}}(S) = k$, partition $\mathcal{P}_k$ contains all the input sequences that generate S and can thus compute S's frequency exactly. To determine $F_k(\mathcal{P}_k)$, we could run any sequential FSM algorithm and filter out all non-pivot sequences afterwards.

Fig. 3 depicts item-based partitioning for our running example with $\sigma = 2$. The center column of the figure shows the input sequences $T \in \mathscr{D}_{\mathrm{ex}}$ (left) along with their candidate subsequences $G_{\pi_{\mathrm{ex}}}(T)$ (right). Pivot items are underlined and candidate subsequences that contain infrequent items are crossed out. The left column illustrates item-based partitioning with sequence representation. For example, $K^{\mathrm{ip}}(T_1) = \{ a_1, c \}$. Consequently, we send T_1 to partitions $\mathcal{P}_{a_1}$ and $\mathcal{P}_c$.

The key idea of [6], [22] is to set $\rho_k(T) = T'$, where T' is a "rewritten" variant of T such that T' is shorter than T but $G_\pi^\sigma(T)$ and $G_\pi^\sigma(T')$ nevertheless agree on the set of pivot sequences for k (but may otherwise be different). This approach is effective in reducing communication costs and it speeds up local mining because partitions are smaller. A second significant performance improvement is to make the local miner F_k aware of the fact that only pivot sequences need to be output [6]. The aforementioned methods have outperformed alternative methods in the experiments of [6], [22]. Their main drawback, however, is that they are suitable for length and gap constraints only. In Sec. V, we discuss how and to what extent these ideas can be lifted to support more powerful subsequence constraints.

Candidate representation. An alternative communication strategy is to send each candidate subsequence to its respective partition. Specifically, we use $K^{\mathrm{ip}}(T)$ as defined above and set $\rho_k(T) = \{ S \mid S \in G_\pi^\sigma(T),\, \kappa^{\mathrm{ip}}(S) = k \}$. Then, $\mathcal{P}_k$ contains as many "copies" of each pivot sequence S as there are sequences in $\mathscr{D}$ that generate S. In other words, we can obtain $F_k(\mathcal{P}_k)$ by simply counting the number of occurrences of each sequence in $\mathcal{P}_k$. Thus, the approach is closely related to SEMI-NAÏVE (the same amount of data is communicated) and suffers from similar drawbacks. Our D-CAND algorithm alleviates these drawbacks via suitable compression techniques. In Sec. VI, we study how far candidate representation can be pushed and how it compares to sequence representation.

The right column of Fig. 3 depicts the use of candidate representation for our running example. For instance, we split $G_{\pi_{\mathrm{ex}}}^2(T_1)$ into two parts $\rho_{a_1}(T_1)$ and $\rho_c(T_1)$, which contain the candidate subsequences with pivot item a_1 and c, respectively, and are sent to the corresponding partitions.

Discussion. Generally, we expect candidate representation to reduce communication cost if few candidates are generated (or can be well compressed) and input sequences are long (and cannot be well compressed). Sequence representation is beneficial if short sequences generate many candidates (as in our running example) that cannot be well compressed. In the remainder of this paper, we derive efficient algorithms based on sequence representation (Sec. V) and candidate representation (Sec. VI) and discuss and evaluate their performance.

IV. DESQ Subsequence Constraints

To develop efficient parallel mining algorithms, we need to be able to peek into the subsequence predicate π. For example, a naïve approach to determine the set of partitioning keys $K^{\mathrm{ip}}(T)$ in item-based partitioning is to first compute and then iterate over $G_\pi(T)$. As $G_\pi(T)$ can be exponential

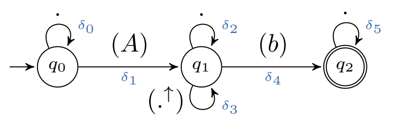


Fig. 4: FST for subsequence predicate π_{ex}.

in the length of T, such an approach is often inefficient. In the remainder of this paper, we adopt the computational model of DESQ [5], [7] for subsequence predicates: it allows to model flexible subsequence constraints (see Fig. 1), yet imposes enough structure to enable efficient mining. In what follows, we summarize relevant concepts of DESQ.

DESQ describes a subsequence predicate π via a compressed *finite state transducer* (FST), which "translates" an input sequence T to its candidate subsequences $G_\pi(T)$. We define an FST as a 6-tuple $(Q, q_S, Q_F, \Sigma, 2^\Sigma \cup \{\epsilon\}, \Delta)$, consisting of a set of states Q, an initial state $q_S \in Q$, a set of final states $Q_F \subseteq Q$, input alphabet Σ, output alphabet $2^\Sigma \cup \{\epsilon\}$, and a transition relation $\Delta \subseteq Q \times 2^\Sigma \times (\Sigma \to 2^\Sigma \cup \{\epsilon\}) \times Q$. Fig. 4 depicts an FST corresponding to subsequence predicate π_{ex}.

A *transition* $\delta \in \Delta$ is a tuple $(q_{\text{from}}, \text{in}_\delta, \text{out}_\delta, q_{\text{to}})$, where q_{from} and q_{to} refer to the source and the target state, $\text{in}_\delta \subseteq \Sigma$ to the set of acceptable input items, and $\text{out}_\delta : \Sigma \to 2^\Sigma \cup \{\epsilon\}$ to a function that computes a set of output items for one accepted input item. Intuitively, transition δ *matches* an input item t if $t \in \text{in}_\delta$ and then (conceptually) non-deterministically produces one of the output items in $\text{out}_\delta(t)$. We require throughout that $\text{out}_\delta(t) \subseteq \text{anc}(t) \cup \{\epsilon\}$, i.e., when a transition outputs an item, it is guaranteed to be an ancestor of its input (which includes the input itself). We use pattern expressions to specify combinations of in_δ and out_δ compactly, see Tab. I (page 2) for some of the supported ones. For example, transition δ_1 in Fig. 4 (labeled (A)) matches any descendant of A and outputs the input item that was matched.

We *simulate* an FST on an input sequence T to find *accepting runs*. Let $T = t_1 t_2 \ldots t_n$. A *run* for T is a sequence $r = \delta_1–\delta_2–\cdots–\delta_n$ of transitions $\delta_i = (q_i, \text{in}_{\delta_i}, \text{out}_{\delta_i}, q'_i)$ such that $q_1 = q_S$, $q_{i+1} = q'_i$ for $1 \le i < n$, and $t_i \in \text{in}_{\delta_i}$ for $1 \le i \le n$. A run is an *accepting run* if $q'_n \in Q_F$. We denote the set of accepting runs for an input sequence T as $R(T)$. For T_5 of $\mathscr{D}_{\text{ex}}$, a (non-accepting) run is $\delta_1–\delta_3–\delta_2$. The accepting runs are $r_1 = \delta_0–\delta_1–\delta_4$, $r_2 = \delta_1–\delta_2–\delta_4$, and $r_3 = \delta_1–\delta_3–\delta_4$.

The accepting runs for T generate the candidate subsequences $G_\pi(T)$. Each accepting run $r \in R(T)$ produces a sequence of *output sets* (sets of output items) $\text{out}_{\delta_1}(t_1)$–$\text{out}_{\delta_2}(t_2)$ –$\cdots$–$\text{out}_{\delta_n}(t_n)$. For example, run r_3 (described above) produces $\{a_1\}$–$\{a_1, A\}$–$\{b\}$. We associate each run r with a set $G_\pi(r)$ of candidate subsequences by taking the Cartesian product of the output sets (and concatenating each resulting tuple). For instance, $G_{\pi_{\text{ex}}}(r_3) = \{a_1\} \times \{a_1, A\} \times \{b\} = \{a_1 a_1 b, a_1 A b\}$. The complete set of candidate subsequences is then given by $G_\pi(T) = \bigcup_{r \in R(T)} G_\pi(r)$. In our example, $G_{\pi_{\text{ex}}}(r_1) = G_{\pi_{\text{ex}}}(r_2) = \{a_1 b\}$ so that $G_{\pi_{\text{ex}}}(T_5) = \{a_1 b, a_1 a_1 b, a_1 A b\}$, as desired.

V. Sequence Representation

In this section, we describe D-SEQ, an algorithm based on sequence representation, i.e., it communicates (potentially rewritten) input sequences to partitions. D-SEQ is aimed at subsequence constraints that produce large numbers of candidate subsequences. We describe efficient methods to find pivot items (Sec. V-A), rewrite the input sequence (Sec. V-B), and mine each partition locally (Sec. V-C).

In what follows, we consider an arbitrary subsequence predicate π, represented by FST $(Q, q_S, Q_F, \Sigma, 2^\Sigma \cup \{\epsilon\}, \Delta)$. For brevity, we write $\kappa(S)$ for $\kappa^{\text{ip}}(S)$ and $K(T)$ for $K^{\text{ip}}(T)$.

A. Pivot Search

For each input sequence T, we aim to determine $K(T)$—the set of pivot items of partitions for which T is relevant (see Eq. (1))—efficiently (line 2 in Alg. 1). For example, for T_1, we aim to determine $K(T_1) = \{a_1, c\}$.

Naïvely, one can generate all candidate subsequences $G_\pi(T)$ and determine the pivot item of each candidate subsequence in the set. However, this approach is often infeasible due to the exponential number of candidate subsequences.

In the DESQ model, there are two sources for the exponential number of candidate subsequences: (1) the Cartesian product can produce exponentially many candidate subsequences for one accepting run and (2) there can be exponentially many accepting runs. In the following, we address both of these causes. We propose an algorithm that, for a given FST, is linear in the length $|T|$ of the input sequence T.

Pivot items of a run. We define $K(r) \subseteq K(T)$ as the pivot items of a run r:

$$K(r) = \{\, k \mid S \in G_\pi(r), \kappa(S) = k \,\}.$$

From DESQ's computational model, it follows $K(T) = \cup_{r \in R(T)} K(r)$. Consequently, we can determine $K(r)$ for each accepting run separately and merge the results.

When is an item in a run a pivot item? First, consider an example run r_4 for a subsequence constraint $\pi' \neq \pi_{\text{ex}}$ with output sets $\{b, c\}$–$\{A\}$–$\{d, a_1\}$. Recall that $b < A < d < a_1 < c$. We have $G_{\pi'}(r_4) = \{ bA\underline{d}, bA\underline{a_1}, \underline{c}Ad, \underline{c}Aa_1 \}$ and, thus, pivots $K(r_4) = \{c, d, a_1\}$. Now consider a general run r with output sets $\text{out}_{\delta_1}(t_1)$–$\text{out}_{\delta_2}(t_2)$–$\cdots$–$\text{out}_{\delta_n}(t_n)$. An item $w \in \text{out}_{\delta_i}(t_i)$ is a pivot item if it is the maximum item of at least one of the candidate subsequences $G_\pi(r)$. The Cartesian product for r produces such a candidate subsequence if there is at least one item $w' \le w$ in every other output set.

In the following, we propose a method to "merge" output sets in linear time. To do so, we investigate further in which cases an item is a pivot item. In a run of length 1 (i.e., with 1 output set), all items are pivot items. For example, r'_4: $\{b, c\}$ produces $G_{\pi'}(r'_4) = \{\underline{b}, \underline{c}\}$. In a run of length 2, an item of one set is a pivot item if it is greater than or equal to the minimum item of the other set. For example, r''_4: $\{b, c\}$–$\{A\}$ produces $G_{\pi'}(r''_4) = \{b\underline{A}, \underline{c}A\}$ (pivots A and c). We have $A \ge \min\{b, c\}$ and $c \ge \min\{A\}$, but $b < \min\{A\}$. In general, we make use of a commutative and associative "pivot merge" function $\oplus$ to determine the pivot items of two output sets U and Q (with $\epsilon < w$ for $w \in \Sigma$):

$$U \oplus Q = \{\, \omega \in U \mid \omega \geq \min(Q) \,\} \cup \{\, \omega \in Q \mid \omega \geq \min(U) \,\}\,.$$

As we have to check for minimum items in every other set, we apply $\oplus$ repeatedly, see Th. 1. For example, we find the pivot items of r_4 as $K(r_4) = \{\, b, c \,\} \oplus \{\, A \,\} \oplus \{\, d, a_1 \,\}$.

Theorem 1: The pivot items $K(r)$ of run r with output sets $\mathrm{out}_{\delta_1}(t_1)$–$\mathrm{out}_{\delta_2}(t_2)$–$\cdots$– $\mathrm{out}_{\delta_{|T|}}(t_{|T|})$ can be computed by

$$K(r) = \mathrm{out}_{\delta_1}(t_1) \oplus \mathrm{out}_{\delta_2}(t_2) \oplus \cdots \oplus \mathrm{out}_{\delta_{|T|}}(t_{|T|}).$$

As there are at most $|\Sigma|$ items in each output set, we can compute $\oplus$ in time $O(\Sigma)$ using appropriate data structures. For a run of length $|T|$, total computation time reduces from $O(|\Sigma|^{|T|})$ via computation of $G_\pi(r)$ to $O(|T||\Sigma|)$ using Th. 1.

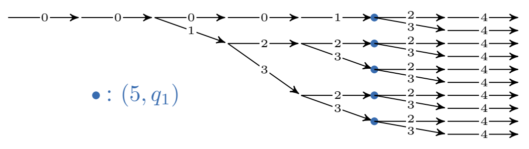
Repeated computation. The number of accepting runs can be exponential in the length $|T|$ of the input sequence: an FST with $|\Delta|$ transitions can have $O(|\Delta|^{|T|})$ accepting runs. Thus, naïvely iterating all runs can be infeasible. Fig. 5a shows all accepting runs for T_2 of our running example, depicted as a trie. Note that there are many more non-accepting runs. Edges are labeled by the number of the FST transition; for example, the uppermost run is δ_0–δ_0–δ_0–δ_0–δ_1–δ_2–δ_4.

As remedy, we propose a dynamic programming approach that is based on the key observation that a *position–state pair* (i, q) of the last-read position i in the input sequence and the current FST state q fully determines the subsequent simulation (from position $i + 1$ to the end of the input sequence). Consequently, we store the result of this subsequent simulation and reuse it when the simulation revisits (i, q). For example, simulation for T_2 visits $(5, q_1)$ multiple times (marked blue in Fig. 5a). Simulation starting in $(5, q_1)$ consistently leads to two accepting runs: $\cdots$–δ_2–δ_4 and $\cdots$–δ_3–δ_4.

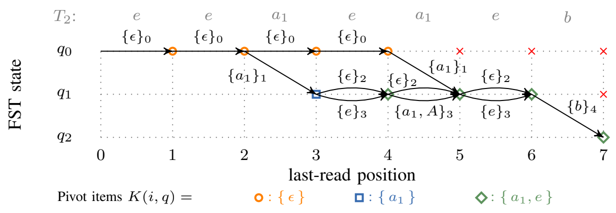
To exploit this property, we interpret FST simulation in a 2-dimensional *position–state grid*. Fig. 5b shows such a grid for T_2. We construct this grid during FST simulation. When first visiting a coordinate pair (i, q), we store the result of the FST simulation starting from (i, q). If the simulation finds accepting runs starting from (i, q), we add the taken transitions into the grid between the corresponding coordinates, labeling them with the number of the taken transition. If the simulation finds no accepting runs starting from (i, q), we mark (i, q) as a *dead end* (red crosses in Fig. 5b).

In Fig. 5b, we label edges by the produced output set and, as subscript, the transition number. For example, the edge leaving $(6, q_1)$ corresponds to FST transition δ_4 and produces $\mathrm{out}_{\delta_4}(b) = \{\, b \,\}$, so we label it $\{\, b \,\}_4$.

We find the pivot items $K(T)$ from the accepting runs, i.e., runs ending in position–state pairs $(|T|, q)$ with $q \in Q_F$. For each position–state pair, we define as $R(i, q)$ a set of *partial runs*: the first i transitions of the accepting runs whose ith transition ends in q. More formally: $R(i, q) = \{\, \delta_1$–$\cdots$–$\delta_i \mid \delta_1$–$\cdots$–δ_i–$\cdots$–$\delta_{|T|} \in R(T),\ \delta_i$ ends in $q \,\}$. For example, $R(4, q_1) = \{\, \delta_0$–$\delta_0$–$\delta_1$–$\delta_2,\ \delta_0$–$\delta_0$–$\delta_1$–$\delta_3 \,\}$. We further define as $K(i, q)$ the pivot items of these partial runs: $K(i, q) = \cup_{r \in R(i,q)} K(r)$. For instance, we have $K(4, q_1) = \{\, a_1, e \,\}$



(a) As trie. Edges are labeled by the transition number.



(b) As position–state grid. Edges are labeled by the produced output set and, as subscript, the transition number.

Fig. 5: Accepting runs for T_2.

(the partial outputs of $(4, q_1)$ are subsequences a_1 and a_1e). In Fig. 5b, we give the set $K(i, q)$ for each coordinate that is part of an accepting run. We get $K(T) = \cup_{q \in Q_F} K(|T|, q)$. Only q_2 is a final state, so $K(T_2) = \{\, a_1, e \,\}$ in our example.

We compute all $K(i, q)$ efficiently in one forward pass over the grid after FST simulation has finished. Denote as $\mathrm{inc}_{(i,q)}$ the incoming transitions of coordinate (i, q), comprising tuples of source state and transition. We compute $K(i, q) = \cup_{(q',\delta) \in \mathrm{inc}_{(i,q)}} K(i-1, q') \oplus \mathrm{out}_\delta(t_i)$. Intuitively, for each incoming transition, we take the pivots of the partial runs up to $i - 1$ and combine them with the output set produced by the incoming transition. For example, we have $\mathrm{inc}_{(4,q_1)} = \{\, (q_1, \delta_2), (q_1, \delta_3) \,\}$ and compute $K(4, q_1) = (\{\, a_1 \,\} \oplus \{\, \epsilon \,\}) \cup (\{\, a_1 \,\} \oplus \{e\}) = \{\, a_1 \,\} \cup \{\, e \,\}$.

In our implementation, we exclude infrequent items (which cannot be pivot items) early on—that is, we do not add any item w with $f(w, \mathscr{D}) < \sigma$ to any set $K(i, q)$. In the example, with $\sigma = 2$, we exclude e.

Using the grid bounds run time polynomially. Directly processing the output sets of $O(|\Delta|^{|T|})$ accepting runs takes $O(|T||\Delta|^{|T|})$ time. Using the grid, we process each coordinate at most once and, since there are at most $|\Delta|$ outgoing transitions per coordinate, obtain a lower computational cost of $O(|T||Q||\Delta|)$.

B. Representation

Previous work [6], [22] introduced ideas to send (shorter) rewritten *variants* of an input sequence T to partitions. This can reduce communication cost and speed up local mining. However, existing ideas focus on and are limited to length and gap constraints. In this section, we lift these ideas to the general case of flexible subsequence constraints.

When constructing $\rho_k(T)$, we aim to drop positions of T that are *irrelevant* for a partition. A position is irrelevant for a partition $\mathcal{P}_k$ if T and the variant of T without the item at this position agree on the set of pivot subsequences for pivot k. For example, all positions with e's of T_2 are irrelevant for pivot

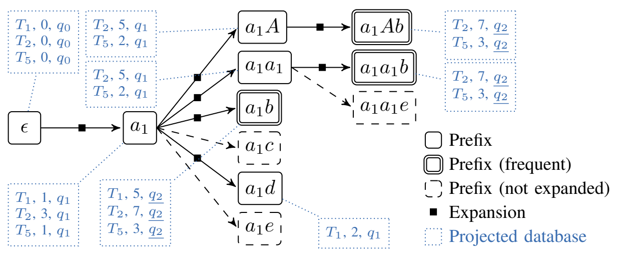


Fig. 6: Search tree for the local mining of partition $\mathcal{P}_{a_1}$ with $\sigma = 2$. Final state q_2 is underlined.

a_1, because $G^2_{\pi_{\text{ex}}}(a_1a_1b)$ and $G^2_{\pi_{\text{ex}}}(T_2)$ agree on the pivot sequences for pivot a_1: $\{\, a_1a_1b, a_1Ab, a_1b \,\}$ (see also Fig. 3).

Naïvely, we can check relevancy by simulating the FST for T and its variant and subsequently compare the set of pivot sequences. However, doing so for all positions and all pivot items is inefficient.

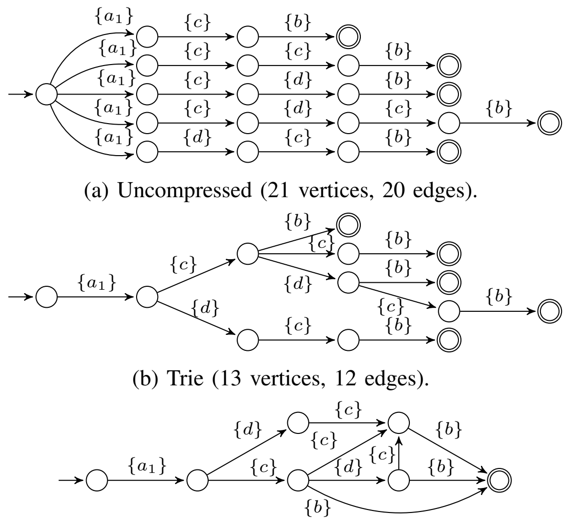
We thus focus on a subset of irrelevant positions that we can identify efficiently: leading and trailing irrelevant positions. That is, we identify the first relevant position and the last relevant position for each pivot item k. We then omit the positions outside this range from $\rho_k(T)$. This ensures that dropping positions does not introduce additional accepting runs (and, thus, additional pivot sequences). The first (last) relevant position is the first (last) position (starting at the beginning of T) that either (1) produces output for a pivot sequence or (2) causes the FST simulation to change to another state of the FST in any accepting run for pivot k. We can identify these positions efficiently in the forward pass over the grid. In our example, for pivot a_1, we find the two irrelevant positions at the beginning of T_2 and, thus, send $\rho_{a_1}(T_2) = a_1ea_1eb$ to partition $\mathcal{P}_{a_1}$.

This sufficient condition worked well in our experiments. We experimented with more sophisticated tests, but they took more time to compute irrelevant positions than they saved in communication and mining. In fact, when patterns occur locally in the input sequence (as is often the case), our sufficient condition already identifies most irrelevant positions.

C. Local Mining

In the following, we discuss how to mine efficiently for frequent subsequences with pivot item k in a partition $\mathcal{P}_k$ (line 7 in Alg. 1). In principle, we can run any FSM algorithm that supports flexible subsequence constraints and discard frequent subsequences S with $\kappa(S) \neq k$. However, in doing so, we may spend a significant amount of time to mine such non-pivot frequent subsequences. Instead, we adapt the DESQ-DFS algorithm, a pattern-growth approach, to mine only pivot sequences. DESQ-DFS was shown to outperform other approaches [5].

Mining starts with the empty sequence and expands this *prefix* recursively by one item at a time, creating a *search tree*. Fig. 6 shows this tree for partition $\mathcal{P}_{a_1}$ of our example. Each node in the tree is associated with a *projected database*, which



(a) Uncompressed (21 vertices, 20 edges).

(b) Trie (13 vertices, 12 edges).

(c) Minimized (7 vertices, 10 edges).

Fig. 7: NFAs for $\rho_c(T_1)$. D-CAND communicates minimized NFAs as shown in (c).

stores a list of 3-tuples (T, i, q) that can produce this prefix—comprising an input sequence T, the last-read position i of T, and the current state q in FST simulation.

At partition $\mathcal{P}_k$, frequent subsequences cannot contain any items $w > k$: such sequences have pivot item $\kappa(S) > k$. Consequently, we do not expand nodes in the search tree with items $w > k$. For example, at partition $\mathcal{P}_{a_1}$, we do not expand the prefix with items c or e, because $e > c > a_1$.

Early stopping. This approach may still produce sequences that consist solely of items $w < k$ and, consequently, have pivot "smaller" than k. In our example, this is not the case for $\mathcal{P}_{a_1}$. In $\mathcal{P}_c$, however, mining yields frequent sequence a_1b, although $\kappa(a_1b) = a_1 < c$. We employ a heuristic to prevent some branches of the search tree that produce such frequent subsequences: for each input sequence T, we determine the last position of T that can potentially produce the pivot item. We then do not use T to expand a prefix that does not contain the pivot item beyond this position.

VI. Candidate Representation

The D-CAND algorithm is based on candidate representation. It is targeted at subsequence constraints that produce small numbers of candidate subsequences. We describe efficient methods to determine pivot items and to construct a compressed representation simultaneously (Sec. VI-A) and to mine directly on this representation (Sec. VI-B).

A. Pivot Search and Representation

In candidate representation, we send to partition $\mathcal{P}_k$ the set of candidate subsequences of T that have pivot item k. Naïvely, we can send a list of these. For instance, for T_1, we can send $\rho_c(T_1) = \{\, a_1cdcb, a_1cdb, a_1cb, a_1dcb, a_1ccb \,\}$ to $\mathcal{P}_c$ and $\rho_{a_1}(T_1) = \{\, a_1db, a_1b \,\}$ to $\mathcal{P}_{a_1}$. However, this approach suffers from similar drawbacks as NAÏVE and SEMI-NAÏVE.

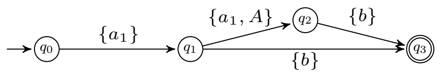


Fig. 8: NFA for $\rho_{a_1}(T_5)$.

Instead, we propose to send a compressed representation and mine on this representation directly. Compression is effective when the candidate subsequences have shared structure. E.g., in $\rho_c(T_1)$, all subsequences start with a_1 and end with b.

We use *nondeterministic finite automata* (NFAs) to compress sets of candidate subsequences. We interpret a set $\rho_k(T)$ as a finite language and construct an NFA for it. That is, this NFA accepts precisely the subsequences in $\rho_k(T)$. For example, Fig. 7a shows such an NFA for $\rho_c(T_1)$. One NFA edge corresponds to an output set $out_{\delta_i}(t_i)$. We can compress an NFA in any way, given that the compressed version also accepts precisely the candidate subsequences in $\rho_k(T)$.

We aim to construct a representation that minimizes shuffle size. Thus, ideally, we want to minimize the NFA. In general, NFA minimization is PSPACE-complete [18]. *Tries*, however, are acyclic, so they can be minimized in linear time [25]. We construct tries and subsequently minimize the tries.

Construction. To construct the NFAs for input sequence T, we simulate the FST to find accepting runs and maintain one trie for each found pivot item $k \in K(T)$. When the simulation finds an accepting run $r \in R(T)$, we insert this run into the tries for $k \in K(r)$. We drop all items $w > k$ from the output sets of the run, because these items produce candidate subsequences with $\kappa(S) > k$. After FST simulation, we minimize the constructed tries. In our example, for T_1, we construct two NFAs: one for $\rho_c(T_1)$ (Fig. 7b shows the trie, Fig. 7c the minimized NFA) and one for $\rho_{a_1}(T_1)$. We do not use a position–state grid to construct NFAs, because we found that the subsequence constraints that D-CAND is aimed at (i.e., constraints that produce only few candidate subsequences) do not benefit significantly from using a grid.

Serialization. We serialize an NFA by writing out information about each of its transitions. Naïvely, for each transition, we write its source state, its label (an output set), and its target state. Additionally, we note which states are final ($\mathcal{F}$). Consider the example NFA for $\rho_{a_1}(T_5)$, depicted in Fig. 8. We write $0\ a_1\ 1 \mid 1\ a_1A\ 2 \mid 2\ b\ 3 \mid 1\ b\ 3 \mid \mathcal{F}: 3$.

To decrease serialization size, we use the following compression: (1) if no source state is given for a transition, the transition starts in the target state of the previous transition and (2) if no target state is given, it ends in a new one.

To obtain the compressed representation, we run *depth-first search* on the NFA and serialize the transitions in the visit order. We write out the label for every transition and additionally (1) the source state if the state was visited before, (2) the target state if the target state was visited before, and (3) a "final" marker if the target state is final and was not visited before. For example, for the NFA for $\rho_{a_1}(T_5)$ and a DFS order $q_0 \to q_1$, $q_1 \to q_2$, $q_2 \to q_3$, $q_1 \to q_3$, we serialize $a_1 \mid a_1A \mid b\mathcal{F} \mid 1\ b\ 3$.

TABLE II: Dataset and hierarchy characteristics.

	NYT	AMZN	AMZN-F	CW50
Total sequences (M)	50	21	21	567
Total items (M)	1 130	83	83	10 774
Unique items[a] (M)	8	10	10	23
Max. sequence length	21 000	44 557	44 557	20 993
Mean sequence length	22.8	3.9	3.9	19.0
Hierarchy items	9 874 089	9 903 422	9 894 624	22 642 566
Max. ancestors	3	282	10	1
Mean ancestors	2.8	5.1	3.5	1.0

We experimented with different serialization schemes and found this one to be the most efficient—considering time for serialization, communication, and deserialization.

Aggregation. We found that different input sequences often send identical NFAs, especially to partitions of frequent items. Therefore, we use a MapReduce *combine* function to aggregate identical NFAs into a single NFA weighted by its frequency.

B. Local Mining

In D-CAND, the most time-consuming computations—that is, simulating the FST on the input sequences—are run in the map phase. Local mining merely counts the number of occurrences of each candidate subsequence in the set of received (weighted) NFAs. To count occurrences efficiently, we operate directly on the compressed NFAs and employ a pattern-growth approach [24].

VII. Experiments

We studied the performance of D-SEQ and D-CAND using multiple real-world datasets and a variety of subsequence constraints. We compared the presented algorithms with each other and to the state-of-the-art w.r.t. performance and scalability. We further compared sequence and candidate representation, and we investigated the benefits of specific components of D-SEQ and D-CAND. Our major insights are:

- Both sequence and candidate representation can reduce communication cost significantly (up to 100x).
- Both D-SEQ and D-CAND scaled nearly linearly with the size of the sequence database.
- D-CAND was up to 5x faster than D-SEQ on selective subsequence constraints (i.e., constraints that select only few candidate subsequences per input sequence). D-SEQ was more robust for less selective constraints.
- D-SEQ and D-CAND exhibited acceptable generalization overhead over existing, specialized methods (usually within 0.9x and 4.3x).

A. Setup

Implementation and cluster. We implemented our algorithms in Java (JDK 1.8) and Scala (version 2.11.8) for Apache Spark (version 2.0.1). Our source code is available online.[2] We

[2]At https://github.com/rgemulla/desq/tree/distributed.

TABLE III: Example subsequence constraints with examples for found frequent sequences. Adapted from [5].

Notation	Subsequence constraint and pattern expression	Example frequent sequences (with support)
Text Mining		
$N_1(\sigma)$	Rel. phr. betw. entities: ENTITY (VERB$^+$ NOUN$^+$? PREP?) ENTITY	*NYT*: *lives in* (4 322), *graduated from* (3 693), *is survived by* (1 749)
$N_2(\sigma)$	Typed rel. phr.: (ENTITY$^\uparrow$ VERB$^+$ NOUN$^+$? PREP? ENTITY$^\uparrow$)	*NYT*: *ORG is offering ENTITY* (2 239), *PER was born in LOC* (11 581)
$N_3(\sigma)$	Copular rel. for an entity: (ENTITY$^\uparrow$ $be^{\uparrow}_{=}$) DET? (ADV? ADJ? NOUN)	*NYT*: *PER be professor* (1 582), *LOC be great place* (99)
$N_4(\sigma)$	Generalized 3-grams before a noun: $(.^\uparrow)\{3\}$ NOUN	*NYT*: *NOUN PREP DET* (8 163 372), *DET ADV ADJ* (760 714)
$N_5(\sigma)$	3-grams, one item generalized: $([.^\uparrow\ .\ .] \mid\mid [.\ .^\uparrow.] \mid\mid [.\ .\ .^\uparrow])$	*NYT*: *who VERB also* (22 223), *human rights NOUN* (21 883)
Recommendation		
$A_1(\sigma)$	Max. 5 electronic items, max. gap 2: $(\mathit{Electr}^\uparrow)[.\{0,2\}(\mathit{Electr}^\uparrow)]\{1,4\}$	*AMZN*: *'MP3 Players' 'Headph.'* (11 761), *'Mice' 'Keyb.' 'Accessib.'* (875)
$A_2(\sigma)$	Sequences of books: $(\mathit{Book})[.\{0,2\}(\mathit{Book})]\{1,4\}$	*AMZN*: *'A Storm of Swords' 'A Feast for Crows'* (153)
$A_3(\sigma)$	Gen. items after a digital camera: $\mathit{DigitalCamera}[.\{0,3\}(.^\uparrow)]\{1,4\}$	*AMZN*: *'Lenses' 'Tripods'* (158), *'Batteries' 'SD&SDHC Cards'* (149)
$A_4(\sigma)$	Musical instruments: $(\mathit{MusicInstr}^\uparrow)[.\{0,2\}(\mathit{MusicInstr}^\uparrow)]\{1,4\}$	*AMZN*: *'MusicInstr' 'Bags&Cases'* (2 158)
Traditional constraints		
$T_1(\sigma,\lambda)$	PrefixSpan: max. length λ: $(.)\big[.^*(.)\big]\{,\lambda\text{-}1\}$	*AMZN* [λ=5]: *'Kindle Fire' 'Folio Case'* (715), *'Subw. Surf.' 'Flappy W.'* (579)
$T_2(\sigma,\gamma,\lambda)$	MG-FSM: max. length λ, max. gap γ: $(.)\big[.\{0,\gamma\}(.)\big]\{1,\lambda\text{-}1\}$	*NYT* [γ=1,λ=5]: *most of the* (115 243), *spoke on cond. anon.* (9 995)
$T_3(\sigma,\gamma,\lambda)$	LASH: max. length λ, max. gap γ, hierarchy: $(.^\uparrow)\big[.\{0,\gamma\}(.^\uparrow)\big]\{1,\lambda\text{-}1\}$	*AMZN-F* [γ=1,λ=5]: *'Pop CD' 'Pop CD' 'Pop CD'* (49 139)

used the authors' implementations of LASH[3] and DESQ[4] and PrefixSpan of Spark 2.0.1. We used a local cluster of 8+1 Dell PowerEdge R720 computers, running CentOS Linux 7.3.1611 and connected with 10 GBit Ethernet. Each of the 8 worker nodes was equipped with two Intel Xeon E5-2640 v2 8-core CPUs, 128 GB of main memory, and four 2 TB NL-SAS 7200 RPM hard disks, the master node had 1 such CPU and 64 GB of memory. The algorithms read input sequences from HDFS (Hadoop 2.5.0) and stored found frequent sequences to HDFS. We ran 1 *executor* with 8 virtual CPU cores and 64 GB of memory per worker node.

Measures. We report end-to-end *run times* as measured by Apache Spark. We report Spark's `shuffleWriteBytes` metric for the map stage as *shuffle size*. All reported measurements are the mean of three independent runs that we ran with no other applications running on the cluster.

Datasets. Tab. II depicts statistics about the used datasets. In the *New York Times Annotated Corpus* (*NYT*) words generalize to their lemma and to their part-of-speech tag. Named entities generalize to their type and to ENTITY. We interpreted one sentence as one input sequence.

The *AMZN* dataset comprises product reviews of Amazon customers [20]. We interpreted the products reviewed by one customer as one input sequence. Items generalize to broader categories and to departments, according to the Amazon product hierarchy. We constructed *AMZN-F*, a variant of AMZN, for algorithms that support only hierarchies of forest form (i.e., each item can generalize to at most one other item): for an item that generalizes to more than one other item, we retained only the generalization to the most frequent parent item. Subsequently, we removed hierarchy items that have only one child when this child has identical item frequency.

The dataset *CW50* is a 50% sample of the ClueWeb09-T09B subset of ClueWeb. We interpreted one sentence as one input sequence. We used no hierarchy for this dataset.

[3]From https://github.com/uma-pi1/lash. LASH is not available for Spark, so we used the authors' Hadoop implementation. Thus, we compare two systems. We argue that a comparison is meaningful nevertheless, as the compared algorithms are compute-bound and run only one round of communication.

[4]From https://github.com/rgemulla/desq/tree/master.

Preprocessing—that is, computing item frequencies and converting the dataset to a frequency-based encoding—took approximately 2 minutes and 10 seconds for both NYT and AMZN and roughly 9 minutes for CW50. As the preprocessing has to be run only once per dataset, we do not include preprocessing times in our experiments.

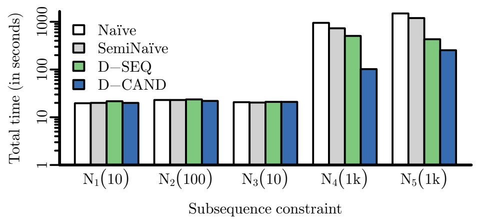
Subsequence constraints. Tab. III depicts the constraints we used in our experiments and example frequent sequences for each constraint. N_1–N_5 are text mining applications, based on [12], [23], [30]. They exploit the NYT hierarchy to specify item constraints. A_1–A_4 are examples for order-aware recommendation tasks. T_1, T_2, and T_3 model the constraint types of existing scalable algorithms. The examples in Tab. III show that flexible constraints allow analysts to increase the usefulness of frequent sequences, e.g., contrast T_2 with N_2.

B. D-SEQ and D-CAND

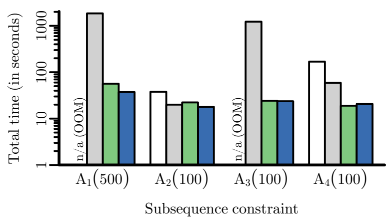
We found that (1) D-SEQ and D-CAND outperformed naïve methods by up to 50x, (2) both sending rewritten input sequences and sending NFA-encoded candidate subsequences lead to compact representations, (3) D-CAND mined flexible subsequence constraints up to 5x faster than D-SEQ, and (4) proposed enhancements to D-SEQ and D-CAND improve performance with minimal overhead.

D-SEQ and D-CAND outperformed naïve methods by up to 50x, see Fig. 9a and Fig. 9b. Existing scalable methods do not support these subsequence constraints.

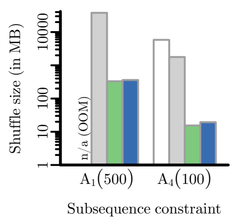
CSPI. The differences in relative performance stem mostly from the *number of candidate subsequences per input sequence* (CSPI) that a subsequence constraint generates. Tab. IV depicts CSPI statistics. We refer to constraints with low CSPI as *selective* and ones with high CSPI as *loose*. The main problem of the naïve methods is that they generate and communicate all candidate subsequences. Consequently, for selective constraints (e.g., $N_1(10)$ or $N_2(100)$), naïve approaches worked relatively well. For greater CSPI (e.g. $N_5(1\text{k})$ or $A_1(500)$), D-SEQ and D-CAND outperformed naïve methods clearly. For loose constraints (with still greater CSPI, e.g., $T_3(10,1,5)$ or $T_1(400,5)$), naïve methods ran out



(a) On NYT



(b) On AMZN



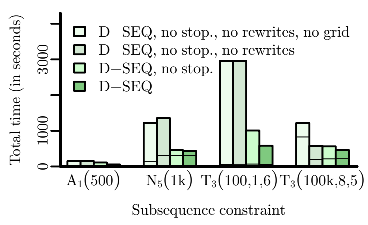
(c) Shuffle size (AMZN)

Fig. 9: Performance for flexible subsequence constraints. Both D-SEQ and D-CAND offer efficient representations and outperform baselines by up to 50x. None of the existing scalable FSM algorithms support these constraints.

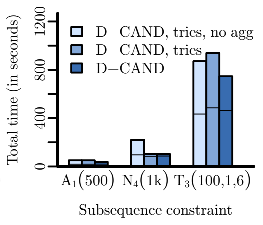
TABLE IV: Statistics on candidate subsequences.

Constraint, dataset	matched seqs. (%)[a]	# cand. seqs. (in million)	CSPI mean	CSPI med.
N_1(10), NYT	3.8	2	1.0	1
N_2(100), NYT	3.8	16	8.5	9
N_3(10), NYT	0.9	1	2.9	3
N_4(1k), NYT	88.5	5 052	115.1	99
N_5(1k), NYT	98.1	6 335	130.2	119
A_1(500), AMZN	5.4	5 050	4 394.4	30
A_2(100), AMZN	5.1	42	38.5	1
A_3(100), AMZN	0.6	3 216	25 716.9	989
A_4(100), AMZN	0.3	205	3 787.6	25
T_3(100,1,5), AMZN-F	47.8	242 309	23 953.1	69
T_3(10,1,5), AMZN-F	48.2	392 492	38 458.2	107
T_1(6400,5), AMZN	5.8	2.5	2.1	
T_1(1600,5), AMZN	17.4	1 830	494.1	
T_1(400, 5), AMZN[b]	37.6	3 235 364	406 146.6	

[a] % of sequences that produce at least one candidate subsequence
[b] Estimated from a 0.1% random sample of the sequences



(a) In D-SEQ: grid, rewrites, and early stopping.



(b) In D-CAND: aggregating and minimizing NFAs.

Fig. 10: Detailed analysis. Each algorithm component accelerates some constraints drastically with little overhead for others. Horizontal lines within bars mark the start of the mine stage.

of memory, because Spark failed to spill accrued shuffle to disk before it exceeded the YARN container memory limit.

Representations. Both sending rewritten input sequences (in D-SEQ) and sending NFA-compressed candidate subsequences (in D-CAND) lead to compact representations. Fig. 9c shows shuffle sizes for two constraints. Both D-SEQ and D-CAND shuffled up to 100 times less data than the naïve methods. In particular, it is notable that the NFA representation in D-CAND is almost as concise as the one of D-SEQ.

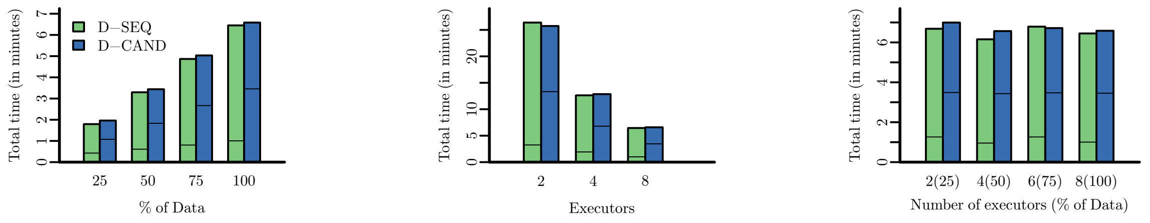
Detailed analysis. In D-SEQ, we studied individually the effects of using the position–state grid, rewriting input sequences, and stopping early, see Fig. 10a. In D-CAND, we studied the effects of aggregating and minimizing NFAs, see Fig. 10b. The effects vary among subsequence constraints. In general, all these enhancements improved performance for some subsequence constraints drastically and added no or only little overhead for the remaining constraints. A horizontal line inside a bar in a figure marks the start of the mine stage.

C. Scalability

We found that our methods scaled nearly linearly with the number of input sequences, achieved significant speed-ups over sequential execution, and were able to mine large datasets, for which sequential execution ran out of memory.

Weak, strong, and data scalability. We observed near-linear scaling of D-SEQ and D-CAND for several subsequence constraints. We report the results for constraint $T_3(100, 1, 5)$. Apart from a constant worker setup time (for creating tasks and broadcasting the dictionary), both map and mine time increased linearly as we increased dataset size (Fig. 11a), decreased linearly as we used more executors (Fig. 11b), and remained roughly constant as we increased both simultaneously (Fig. 11c). To vary dataset size, we created random samples of the AMZN-F dataset with 25%, 50%, and 75% of the original sequences. We adapted σ to the number of sequences in the samples, such that the shuffle size increases proportionally and the number of frequent sequences increases slightly with increasing dataset size. Specifically, for $T_3(\sigma, 1, 5)$, we set σ to 25, 50, 75, and 100, respectively.

Speed-up over sequential execution. Tab. V depicts run times for DESQ-DFS [5], D-SEQ, and D-CAND. DESQ-DFS is a suitable sequential baseline as it, according to [5], outperforms alternative methods. We ran DESQ-DFS on one machine of our cluster with 124 GB of maximum heap memory. We ran our methods using standard settings (i.e., 65 CPU cores in total, including 1 core for the driver). The speed-ups are not perfect as our methods run additional computation to separate the work into independent parts and communicate over network. Distributed execution leads to better speed-ups for longer-running tasks due to constant worker setup time. D-CAND achieves a (high) 58x speed-up for N_4(1k) because it aggregates the many identical NFAs that N_4 produces. DESQ-DFS ran out of memory (OOM) for both CW50 tasks with both 124 GB and 204 GB (using swap) heap space.

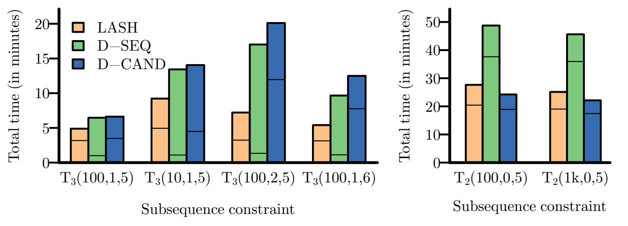


(a) Data scalability (8 executors) (b) Strong scalability (100% of data) (c) Weak scalability

Fig. 11: Scalability. D-SEQ and D-CAND scale nearly linearly. Depicted: T_3(100,1,5) on dataset AMZN-F.

TABLE V: Speed-up over sequential execution. DESQ-DFS runs on 1 CPU core, D-SEQ and D-CAND on 65.

		Run time in minutes (speed-up)		
Constraint	Dataset	DESQ-DFS	D-SEQ	D-CAND
N_4(1k)	NYT	99.37	8.44 (12×)	1.70 (58×)
N_5(1k)	NYT	67.46	7.18 (9×)	4.23 (16×)
T_3(10,1,5)	AMZN-F	280.63	13.43 (21×)	14.05 (20×)
T_3(10k,1,5)	AMZN-F	15.56	2.39 (7×)	2.81 (6×)
T_3(100,3,5)	AMZN-F	676.70	38.19 (18×)	49.11 (14×)
T_2(100,0,5)	CW50	(OOM)	48.73 (n/a)	24.24 (n/a)
T_2(1k,0,5)	CW50	(OOM)	45.62 (n/a)	22.13 (n/a)



(a) AMZN-F (b) CW50

Fig. 12: LASH setting. D-SEQ and D-CAND offer acceptable generalization overhead over the specialized LASH algorithm.

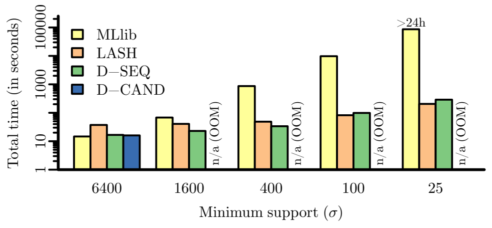


Fig. 13: MLlib setting ($T_1(\sigma, 5)$ on AMZN without hierarchy). D-SEQ is competitive to LASH and outperforms MLlib.

D. *Existing Methods*

Comparing to existing scalable FSM algorithms, we found that D-SEQ, D-CAND, and even naïve methods can mine tasks that existing methods cannot mine efficiently, and that D-SEQ and D-CAND were competitive to existing methods even in their specialist settings. D-CAND can run out of memory for very loose constraints.

Our algorithms support more general and more flexible subsequence constraints than existing scalable FSM algorithms. This allows for more useful output and mining can focus on relevant patterns early on. Existing methods [6], [8], [21], [22] support only a subset of typical constraints and are therefore of limited use for many applications. For example, existing methods cannot mine constraints such as N_1-N_5 or A_1-A_4. In this section, we investigate how our general algorithms perform in the special settings of existing algorithms.

LASH setting. In the setting LASH is optimized for (max. gap and max. length constraints and item hierarchies), D-SEQ and D-CAND were within 0.9x and 2.8x the run times of LASH, see Fig. 12a and Fig. 12b. For LASH, the horizontal line within a bar depicts the end of the last map task. We argue that both D-SEQ (within 1.3x and 2.5x) and D-CAND (within 0.9x and 2.8x) offer acceptable generalization overhead over LASH. As LASH, D-SEQ sends rewritten input sequences to the partitions. In the LASH setting, D-SEQ is slower than LASH because LASH employs rewrite and mining techniques that are specific for its setting and not directly applicable to the more general setting of D-SEQ. To have comparable computational resources for LASH, which is implemented for Apache Hadoop, we ran LASH with 8 map and 8 reduce tasks per worker node and 8 GB of main memory per task.

MLlib setting. D-SEQ outperformed MLlib in its specialized setting (max. length constraint, no item hierarchies, and arbitrary gaps). D-CAND ran out of memory for these very loose constraints, see Fig. 13. LASH can mine this setting, so we included it for reference. We assume that D-SEQ was slightly faster than LASH for short (<50s) tasks because task scheduling takes longer in Hadoop than in Spark, and because Hadoop writes shuffle data to disk between the map and the reduce stage. For longer-running tasks, this difference is negligible. D-CAND ran out of memory while constructing NFAs, as Spark again failed to spill to disk in time. Note however that that the MLlib setting (with arbitrary gaps and no hierarchy) is the worst possible subsequence constraint for D-CAND: the gaps allow for the maximum theoretical number of accepting runs. We argue that the MLlib setting is too loose for most applications.

We omit a separate comparison to MG-FSM and Suffix-σ, as LASH strictly outperformed MG-FSM [6] and MG-FSM outperformed Suffix-σ [4] in other studies.

VIII. Related work

Due to space constraints, we focus our discussion on methods that handle flexible constraints and on distributed methods.

Subsequence constraints. There exist many approaches for constraining which subsequences should be considered for mining. GSP [28] introduced minimum and maximum gap constraints as well as sliding windows for time-annotated sequences. cSpade [34] supports length, gap, and item constraints. Wu et al. [32] studied periodic wild card gaps. Regular expressions as "output filter" were proposed in the SPIRIT family of algorithms [13], RE-Hackle [2], and SMA algorithms [31]. Such filters are evaluated on only the subsequence, but not the input sequence, so that context constraints cannot be specified. DESQ [5], [7] extends regular expressions with contextual constraints by considering both input sequence and subsequence as input for evaluating constraints. Our methods use the DESQ framework to specify and evaluate constraints. However, DESQ is a sequential algorithm and, consequently, does not scale to large datasets.

Scalable mining. To mine large datasets efficiently, parallel algorithms have been developed for shared [35] and distributed memory architectures [15], [16], but without support for constraints or hierarchies. Apache Spark's MLlib library [21] features a distributed version of PrefixSpan [24] for distributed FSM with sequences of itemsets, but without support for hierarchies or subsequence constraints other than maximum subsequence length. It uses *prefix-based partitioning*; that is, it recursively partitions sequences by their first items. Thus, it runs multiple rounds of communication. In the context of itemset mining, Savasare et al. [26] proposed to partition inputs (instead of outputs). Their approach has the drawback that all candidates need to be communicated to all workers.

Most closely related to our work is a group of distributed sequential pattern mining algorithms targeted towards the MapReduce programming model: Suffix-σ [8], MG-FSM [4], [22], and LASH [6]. Suffix-σ mines subsequences of consecutive items in one MapReduce step with suffix-partitioning. However, it does not support gaps. MG-FSM and LASH are distributed FSM algorithms with maximum gap and maximum length constraints. They use item-based partitioning and sequence representation with specialized rewrite techniques. The methods are inspired by item-based partitioning for parallel itemset mining [10], [16]. LASH extends MG-FSM with item hierarchies and introduces a technique to focus local mining on pivot sequences. According to [6], LASH outperforms MG-FSM. MG-FSM and LASH inspired D-SEQ, which is more general. D-SEQ supports many more types of subsequence constraints, including the ones of MG-FSM and LASH.

IX. Conclusion

We described D-SEQ and D-CAND, the first two FSM algorithms that are scalable and support flexible subsequence constraints. We demonstrated that they can mine varied types of subsequence constraints efficiently, scale nearly linearly, and offer acceptable generalization overhead over existing, specialized methods.

Acknowledgment

This work was supported by a Software Campus grant of the German Ministry of Education and Research (01IS17052).

References

[1] R. Agrawal and R. Srikant. Mining sequential patterns. ICDE '95.

[2] H. Albert-Lorincz and J. Boulicaut. Mining frequent sequential patterns under regular expressions: a highly adaptative strategy for pushing constraints. SDM '03.

[3] C. Antunes and A. Oliveira. Inference of sequential association rules guided by context-free grammars. ICGI '02.

[4] K. Beedkar, K. Berberich, R. Gemulla, and I. Miliaraki. Closing the gap: Sequence mining at scale. *TODS*, 40(2):8:1–8:44, 2015.

[5] K. Beedkar and R. Gemulla. DESQ: Frequent sequence mining with subsequence constraints. ICDM '16.

[6] K. Beedkar and R. Gemulla. LASH: Large-scale sequence mining with hierarchies. SIGMOD '15.

[7] K. Beedkar, R. Gemulla, and W. Martens. A unified framework for frequent sequence mining with subsequence constraints. *To appear in TODS*.

[8] K. Berberich and S. Bedathur. Computing N-gram statistics in MapReduce. EDBT '13.

[9] A. Brazma, I. Jonassen, J. Vilo, and E. Ukkonen. Pattern discovery in biosequences. ICGI '98.

[10] G. Buehrer et al. Toward terabyte pattern mining: An architecture-conscious solution. PPoPP '07.

[11] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. OSDI '04.

[12] A. Fader, S. Soderland, and O. Etzioni. Identifying relations for open information extraction. EMNLP '11.

[13] M. Garofalakis, R. Rastogi, and K. Shim. SPIRIT: Sequential pattern mining with regular expression constraints. VLDB '99.

[14] F. Giannotti, M. Nanni, and D. Pedreschi. Efficient mining of temporally annotated sequences. SDM '06.

[15] V. Guralnik, N. Garg, and G. Karypis. Parallel tree projection algorithm for sequence mining. Euro-Par '96.

[16] V. Guralnik and G. Karypis. Parallel tree-projection-based sequence mining algorithms. *Parallel Computing*, 30(4):443–472, 2004.

[17] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. SIGMOD '00.

[18] T. Jiang and B. Ravikumar. Minimal NFA problems are hard. *SIAM Journal on Computing*, 22(6):1117–1141, 1993.

[19] A. Lopez. Statistical machine translation. *ACM Computing Surveys*, 40(3):8:1–8:49, 2008.

[20] J. McAuley, R. Pandey, and J. Leskovec. Inferring networks of substitutable and complementary products. KDD '15.

[21] X. Meng et al. MLlib: Machine learning in Apache Spark. *The Journal of Machine Learning Research*, 17(1):1235–1241, 2016.

[22] I. Miliaraki et al. Mind the gap: Large-scale frequent sequence mining. SIGMOD '13.

[23] N. Nakashole, G. Weikum, and F. Suchanek. PATTY: A taxonomy of relational patterns with semantic types. EMNLP-CoNLL '12.

[24] J. Pei et al. PrefixSpan: Mining sequential patterns efficiently by prefix-projected pattern growth. ICDE '01.

[25] Dominique Revuz. Minimisation of acyclic deterministic automata in linear time. *Theoretical Computer Science*, 92(1):181–189, 1992.

[26] A. Savasere, E. Omiecinski, and S. Navathe. An efficient algorithm for mining association rules in large databases. VLDB '95.

[27] K. Smets and J. Vreeken. Slim: Directly mining descriptive patterns. SDM '12.

[28] R. Srikant and R. Agrawal. Mining sequential patterns: Generalizations and performance improvements. EDBT '96.

[29] J. Srivastava et al. Web usage mining: Discovery and applications of usage patterns from web data. *SIGKDD Explorations*, 1(2):12–23, 2000.

[30] Google Ngram Viewer Team. Google Books Ngram Viewer. https://books.google.com/ngrams/info, 2013. Accessed: 2018-10-10.

[31] R. Trasarti, F. Bonchi, and B. Goethals. Sequence mining automata: A new technique for mining frequent sequences under regular expressions. ICDM '08.

[32] Y. Wu et al. Mining sequential patterns with periodic wildcard gaps. *Applied Intelligence*, 41(1):99–116, 2014.

[33] M. Zaharia et al. Spark : Cluster computing with working sets. HotCloud '10.

[34] M. Zaki. Sequence mining in categorical domains: incorporating constraints. CIKM '00.

[35] M. Zaki. Parallel sequence mining on shared-memory machines. *Journal of Parallel and Distributed Computing*, 61(3):401–426, 2001.

[36] M. Zaki. SPADE: An efficient algorithm for mining frequent sequences. *Machine Learning*, 42(1-2):31–60, 2001.